
Tema 6: Excepciones

Tema 6 – Excepciones

- Introducción
- Gestión de excepciones
- Lanzamiento y declaración de excepciones
- Creación de excepciones propias
- Excepciones encadenadas
- Bloque finally
- Ventajas de la excepciones

Introducción

- En todo programa se producen errores
 - Errores en tiempo de compilación
 - Errores en tiempo de ejecución
- Situación ideal: poder detectar todos los errores en tiempo de compilación
 - Esto no es posible: no todos los errores se pueden detectar en tiempo de compilación (p.e. dividir por una variable cuyo valor es 0 en algunos casos)

Introducción

- Errores en tiempo de compilación
 - Léxicos

```
private String cadena = "hola;
```

String literal is not properly closed by a double-quote

- Sintácticos

```
private numero int = 0;
```

Syntax error on token "int", invalid VariableDeclaratorId

Introducción

- Errores en tiempo de compilación
 - Semánticos

```
private String cadena = new Integer(5) ;
```



Type mismatch: cannot convert from Integer to String

Introducción

- Errores en tiempo de ejecución
 - Recuperables
 - Se pueden detectar y se deben tratar
 - p.e. un timeout en una conexión de red, un fichero que no existe...)
 - No recuperables
 - Errores que, habitualmente, son causados por el sistema
 - p.e. falta de memoria, fallo en la máquina virtual...
 - Son errores graves, que suelen finalizar la ejecución del programa

Introducción

- ¿Qué hacer ante un error en tiempo de ejecución?
 - ¿Ignorarlo?, ¿terminar la ejecución?,
¿hacer algo? ¿mostrar un error al usuario?
- Es necesario poder manejar los errores que se producen en tiempo de ejecución

Introducción

- Los errores en tiempo de ejecución en Java se denominan **excepciones** (**exceptions**)
- Cuando se produce una excepción, se **interrumpe** la ejecución secuencial de las sentencias
- No se ejecuta la sentencia posterior a aquella en la que se produce la excepción
- Por defecto se muestra por pantalla un informe del error producido

Introducción

```
public class Prueba {  
    public static void main(String[] args) {  
  
        System.out.println("Sentencia 1");  
        System.out.println("Sentencia 2");  
  
        //Sentencia 3  
        int divisionPorCero = 4 / 0;  
  
        //Estas sentencias no se ejecutarán  
        System.out.println("Sentencia 4");  
        System.out.println("Sentencia 5");  
    }  
}
```



```
Sentencia 1  
Sentencia 2  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at Prueba.main(Prueba.java:10)
```

Introducción

- Finalizar la ejecución del programa es una medida **demasiado drástica** cuando se produce una excepción
- Es necesario poder ejecutar cierto código cuando se produzca una excepción
 - Informar al usuario con un cuadro de diálogo (Timeout, fichero inexistente)
 - Hacer log del error
- Ejecutar código ante una excepción se denomina **capturar, manejar o gestionar** la excepción

Tema 6 – Excepciones

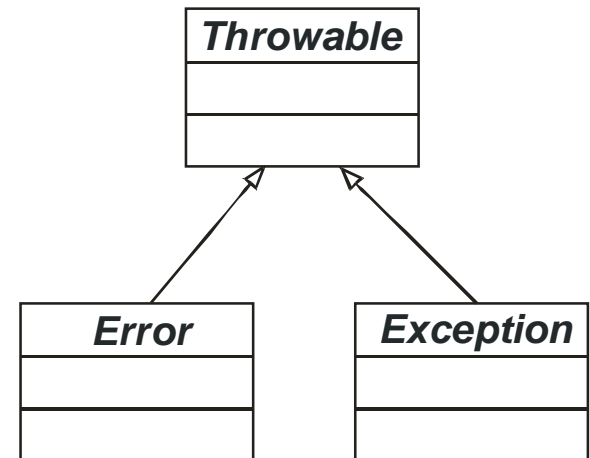
- Introducción
- **Gestión de excepciones**
- Lanzamiento y declaración de excepciones
- Creación de excepciones propias
- Excepciones encadenadas
- Bloque finally
- Ventajas de las excepciones

Gestión de Excepciones

- Siempre que se produce una excepción, se crea un objeto que mantiene información sobre el error producido
- Dependiendo del error, se creará un objeto de una clase determinada
- Usando métodos de los objetos excepción podremos conocer detalles del error producido

Gestión de excepciones

- **`java.lang.Throwable`**: Clase padre de todas las excepciones
- **`java.lang.Error`**: problemas serios que normalmente no se debería intentar manejar (Error de la máquina virtual,...)
- **`java.lang.Exception`**: situaciones que una aplicación podría querer manejar (Error de red, error de acceso a un fichero,...)



Gestión de excepciones

- Clase `java.lang.Error`:
 - Excepciones que indican problemas muy graves
 - Error en la máquina virtual
 - Error con los hilos de ejecución
 - Suelen ser no recuperables
 - No suelen ser capturadas
 - Suelen provocar que se finalice la ejecución del programa

Gestión de excepciones

- Ejemplo de Error
 - Error en la máquina virtual

```
java.lang.Object
    java.lang.Throwable
        java.lang.Error
            java.lang.VirtualMachineError
```

Gestión de excepciones

- Clase `java.lang.exception`:
 - Excepciones que indican problemas razonables que son recuperables
 - Fallo en la red (se desconecta el cable)
 - Fallo de acceso a disco (se saca la llave USB)
 - ...
 - Normalmente deben ser capturadas

Gestión de excepciones

- Ejemplo de **Exception**
 - Se produce cuando se intenta convertir una cadena a entero y no corresponde a un entero

```
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        java.lang.IllegalArgumentException
          java.lang.NumberFormatException
```

Gestión de excepciones

- Algunos métodos de la clase **Throwable**:
 - **public String getMessage()**
 - Devuelve un mensaje que informa de la excepción
 - **public void printStackTrace()**
 - Imprime los métodos que estaban en la pila de ejecución cuando se produjo la excepción por la salida estándar
 - Incluye la línea donde se produce la excepción
 - Es el comportamiento por defecto cuando no se maneja la excepción

Gestión de excepciones

- Para gestionar una excepción hay que indicar el bloque de código donde es posible que se produzca una excepción
 - Bloque **try**
- Hay que indicar que sentencias ejecutar en caso de que en ese código se produzca una excepción
 - Se puede especificar la excepción que queremos tratar
 - Se pueden poner diferentes tratamientos para distintas excepciones
 - Bloques **catch**

Gestión de excepciones

Clase de
excepción
que se
maneja

```
System.out.println("Inicio método");  
  
try {  
    float div = 5/0;  
    System.out.println("Div="+div);  
} catch (ArithmeticException e) {  
    e.printStackTrace();  
}  
  
System.out.println("Fin metodo");
```

Código que
se ejecuta
siempre

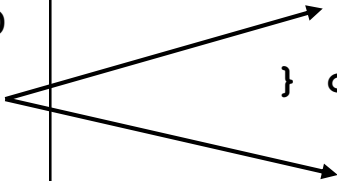
Código que
se ejecuta
si no hay
error

Código que
se ejecuta
si hay error

Gestión de excepciones

```
public class Divisor {  
  
    public static void main(String[] args) {  
  
        try {  
            int numero1 = Integer.parseInt(args[0]);  
            int numero2 = Integer.parseInt(args[1]);  
            float div = numero1 / numero2;  
  
            System.out.println("Div="+div);  
  
        } catch (ArithmeticException e) {  
            System.out.println("Error aritmetico");  
            System.out.println(e.getMessage());  
        } catch (NumberFormatException e){  
            System.out.println("Error de formato");  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

Dependiendo
del error, se
ejecuta un
código u otro



Gestión de excepciones

```
public class Divisor2 {  
    public static void main(String[] args) {  
        try {  
            int numero1 = Integer.parseInt(args[0]);  
            int numero2 = Integer.parseInt(args[1]);  
  
            int div = dividir(numero1, numero2);  
  
            System.out.println("Div=" + div);  
  
        } catch (ArithmeticException e) {  
            System.out.println(e.getMessage());  
        } catch (NumberFormatException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
  
    public static int dividir(int num, int den) {  
        return num / den;  
    }  
}
```

Las excepciones se propagan a través de las llamadas a métodos

La excepción se produce dentro de un método y se captura fuera del método

Puede haber varios métodos entre la excepción y su captura

Gestión de excepciones

- Se pueden poner clases padre en los bloques **catch**
 - Se pueden agrupar varias excepciones en un único **catch** si tienen una clase padre en común
 - El orden de colocación de los **catch** es importante, primero deben ponerse los **catch** de las clases hijas, si no, no se ejecutarán nunca

Gestión de excepciones

```
public class Divisor {  
  
    public static void main(String[] args) {  
  
        try {  
            int numero1 = Integer.parseInt(args[0]);  
            int numero2 = Integer.parseInt(args[1]);  
            float div = numero1 / numero2;  
  
            System.out.println("Div="+div);  
  
        } catch (Exception e) {  
            System.out.println("Error en aplicación");  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

Independientemente de la excepción producida, se ejecuta este manejador porque

ArithmeticException y **NumberFormatException** heredan de **Exception**

Ejercicio 1

- Implementar los ejemplos anteriores
- Probar la gestión de excepciones con diferentes parámetros
- Quitar los manejadores de excepciones y comparar el funcionamiento
- Consultar el JavaDoc de las clases `Exception`, `ArithmeticException`, ...

Tema 6 – Excepciones

- Introducción
- Gestión de excepciones
- Lanzamiento y declaración de excepciones
- Creación de excepciones propias
- Excepciones encadenadas
- Bloque finally
- Ventajas de las excepciones

Lanzamiento y declaración de excepciones

- El desarrollador puede generar o **lanzar** excepciones cuando se producen condiciones inesperadas
- Se crea un objeto de una clase que herede de Exception y se **lanza** o **eleva**
- Se lanza con la sentencia **throw**
- Cuando se lanza una excepción se interrumpe el flujo de ejecución secuencial

Lanzamiento y declaración de excepciones

La
excepción
se crea y
se lanza

```
public class SumaSucesiones {  
    public SumaSucesiones(Sucesion s1, Sucesion s2){  
        if(s1.getNumElementos() != s2.getNumElementos()){  
            throw new IllegalArgumentException(  
                "Deben tener el mismo tamaño");  
        } else {  
            this.s1 = s1;  
            this.s2 = s2;  
        }  
    }  
}
```

La
excepción
se
captura.
Es el
objeto
anterior

```
try {  
    Sucesion s1 = ...  
    Sucesion s2 = ...  
    SucesionSuma ss = new SumaSucesiones(s1,s2);  
} catch (IllegalArgumentException e){  
    System.out.println(e.getMessage());  
}
```

Ejercicio 2

- Implementar el ejemplo anterior
- En el constructor de la clase **SumaSucesiones** comprobar que las dos sucesiones tienen el mismo número de elementos
- Crear un código de prueba que capture la excepción **IllegalArgumentException**

Lanzamiento y declaración de excepciones

- ¿Cómo sabemos las excepciones que se pueden producir en un método?
 - JavaDoc de los métodos
 - `NumberFormatException` en el método `Integer.parseInt(String s)`
 - Especificación del lenguaje Java
 - `NullPointerException`
 - `ClassCastException`
 - `ArithmeticException`

Lanzamiento y declaración de excepciones

- Para algunos tipos de excepciones, el **compilador debe asegurarse** de que el desarrollador sabe que se pueden producir
 - Por ejemplo: Verificación de la URL al crear un objeto de la clase `java.net.URL`
- Estas excepciones no se pueden “ignorar”, es obligatorio o **capturarlas** o indicar que se **pueden lanzar** en el método actual
- Estas excepciones deben estar declaradas en la cabecera de los métodos en los que se pueden lanzar (se pueden ver en JavaDoc)

Lanzamiento y declaración de excepciones

```
public class GestorURLs {  
    ArrayList urls = new ArrayList();  
    public void addURL(String url) {  
        try {  
            urls.add(new URL(url));  
        } catch (MalformedURLException e) {  
            System.out.println("Error en la url");  
        }  
    }  
}
```

Captura de excepción
lanzada en el
constructor de URL

```
public class GestorURLs {  
    ArrayList urls = new ArrayList();  
    public void addURL(String url)  
        throws MalformedURLException  
    {  
        urls.add(new URL(url));  
    }  
}
```

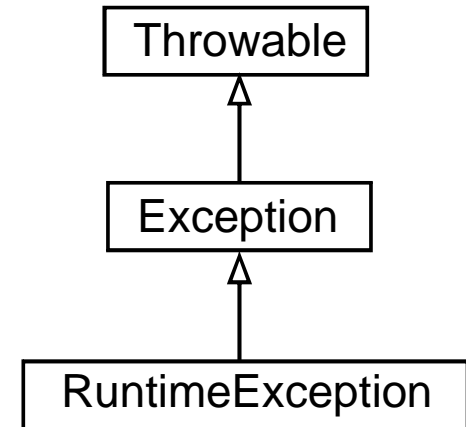
Declaración de que una
excepción del tipo
MalformedURLException
se puede lanzar en este
método. Se pueden poner
varias excepciones

Lanzamiento y declaración de excepciones

- Existen dos tipos de excepciones
 - Excepciones que se pueden capturar o se pueden ignorar
 - Excepciones que no se pueden ignorar (hay que capturarlas o declararlas)

Lanzamiento y declaración de excepciones

- Excepciones que se pueden capturar o se pueden ignorar
 - Son todas las clases hijas de **RuntimeException**
 - Ejemplos
 - **NullPointerException**
 - **ClassCastException**
 - **ArrayIndexOutOfBoundsException**
 - **ArithmeticException**



Lanzamiento y declaración de excepciones

- Excepciones que no se pueden ignorar (hay que capturarlas o declararlas)
 - Cualquier excepción que no sea hija de **RuntimeException**
 - Ejemplos
 - **MalformedURLException**
 - **IOException**
 - **SQLException**
 - **TimeoutException**
 - **ParseException**

Lanzamiento y declaración de excepciones

- Resumen
 - Se puede capturar una excepción proporcionando un manejador para ella (**try** / **catch**)
 - Si una excepción no se puede ignorar (no es **RuntimeException**) y el método no captura la excepción, se debe declarar que en ese método se puede lanzar dicha excepción (**throws**)
 - En un método una excepción se puede lanzar
 - De forma directa creando la excepción (**throw**)
 - De forma indirecta a través de excepciones producidas en los métodos a los que se llama

Ejercicio 3

- Implementa el ejemplo anterior
- Crea la clase `GestorURLs` con el método `addURL(String url)`
- Crea un código de test que use el método `addURL(...)`
- Implementar la captura interna de la excepción y comparar su funcionamiento con el lanzamiento de la excepción (es necesario la declaración de la excepción)

Tema 6 – Excepciones

- Introducción
- Gestión de excepciones
- Lanzamiento y declaración de excepciones
- Creación de excepciones propias
- Excepciones encadenadas
- Bloque finally
- Ventajas de las excepciones

Creación de excepciones propias

- Siempre que se necesite lanzar una excepción, hay que buscar las existentes en la librería de Java
- Si no existe una clase que represente la excepción que se quiere lanzar, se debe crear una clase para la excepción
- El desarrollador debe elegir si la excepción debe heredar de **RuntimeException** (puede ser ignorada)
- Al constructor de **Exception** y **RuntimeException** se le puede pasar un mensaje con **String** que informe de la excepción

Creación de excepciones propias

- Ejemplo de excepción propia

```
public class FullListException extends Exception {  
    public ListFullException(int maximoTamanno){  
        super("Se ha superado el tamaño máximo de "  
            +maximoTamanno);  
    }  
}
```

- Únicamente crea el mensaje de la excepción

Ejercicio 4

- Crear la clase **NumerosPescaderia** que permita controlar un contador de números de una pescadería
- En el constructor se debe indicar un número máximo de números a distribuir
- Debe tener un método **getNumero()** que devuelva “la vez” o eleve una excepción en caso de haber alcanzado el límite máximo
- Hay que crear una clase para la excepción **NoMoreNumbersException**

Tema 6 – Excepciones

- Introducción
- Gestión de excepciones
- Lanzamiento y declaración de excepciones
- Creación de excepciones propias
- Excepciones encadenadas
- Bloque finally
- Ventajas de las excepciones

Excepciones encadenadas

- En algunas ocasiones, las excepciones de alto nivel (Por ejemplo, **SQLException**) son causadas por excepciones de más bajo nivel (Por ejemplo, problemas en la red **IOException**)
- Cuando una excepción de alto nivel ha sido causada por otra excepción de bajo nivel, la excepción de alto nivel guarda una referencia a la excepción de bajo nivel
- Cuando se muestra la traza por pantalla, aparecen todas las excepciones encadenadas

Excepciones encadenadas

- Se puede acceder a la “causa” de una excepción con el método **getCause()**
- Las excepciones tienen constructores para indicar el mensaje y la causa

```
public Integer loadConfigData() throws ConfigFileException {  
    try {  
        String numeroStr = ...  
        return new Integer(numeroStr);  
    } catch (NumberFormatException e) {  
        throw new ConfigFileException  
            ("There is an error un config file", e);  
    }  
}
```

Excepciones encadenadas

- Podemos implementar nuestras excepciones de forma encadenada

```
public class ConfigFileException extends Exception {  
    public ConfigFileException(String msg){  
        super(msg) ;  
    }  
    public ConfigFileException(String msg,  
                                Throwable cause) {  
        super(msg, cause) ;  
    }  
}
```

Excepciones encadenadas

- Implementa el ejemplo anterior
- Implementa el método loadConfigData leyendo un número entero de un fichero de texto
- Implementa la clase ConfigFileException
- Observa la traza cuando se produce una excepción encadenada

Tema 6 – Excepciones

- Introducción
- Gestión de excepciones
- Lanzamiento y declaración de excepciones
- Creación de excepciones propias
- Excepciones encadenadas
- **Bloque finally**
- Ventajas de las excepciones

Bloque finally

- Las excepciones hacen que los métodos finalicen su ejecución en el punto donde se produjo la excepción
- ¿Qué ocurre si un método abre un fichero y se produce una excepción antes de haberlo cerrado? Podemos capturarla y cerrar el fichero
- ¿Y si no queremos capturarla? ¿Si queremos elevarla al método que llama?

Bloque finally

- El bloque **finally** es un bloque opcional al final de la sentencia **try/catch**
- Si existe el bloque **finally**, pueden quitarse todos los **catch** y no se captura la excepción

```
try {  
    <sentencias>  
} catch (Exception e) {  
    <sentencias>  
} finally {  
    <sentencias>  
}
```

```
try {  
    <sentencias>  
} finally {  
    <sentencias>  
}
```

Bloque finally

- El código del bloque **finally** se ejecuta **SIEMPRE**, se produzca o no alguna excepción
- Si se produce alguna excepción, el código del bloque **finally** se ejecuta se capture o no la excepción
- Es un bloque donde se cierran los ficheros, se cierran la conexión con la base de datos, sockets...

Bloque finally

```
public int[] cargarFichero(String nombre) throws IOException,
    NumberFormatException {

    // Abrimos el fichero
    FileInputStream is = new FileInputStream(nombre);

    try {
        // Procesamos el contenido del fichero
        BufferedReader br = new BufferedReader(new InputStreamReader(is));
        String line = br.readLine();
        int numEnteros = Integer.parseInt(line);
        int[] enteros = new int[numEnteros];
        for (int i = 0; i < numEnteros; i++) {
            enteros[i] = Integer.parseInt(br.readLine());
        }
        return enteros;
    } finally {
        //Se generen o no excepciones en el try, el
        //fichero se cierra
        is.close();
    }
}
```

Bloque finally

- Implementar el ejemplo anterior
- El formato del fichero es:
 - Primera línea, número de enteros
 - Resto líneas, un entero en cada línea
- Comprobar que el bloque **finally** se ejecuta siempre, existan o no excepciones

Tema 6 – Excepciones

- Introducción
- Gestión de excepciones
- Lanzamiento y declaración de excepciones
- Creación de excepciones propias
- Excepciones encadenadas
- Bloque finally
- Ventajas de las excepciones

Ventajas de las excepciones

- Antes de las excepciones, se informaba de los errores devolviendo un entero con el código de error
- ¿Cómo devolver un objeto o el código de error?
- La programación se complicaba porque el código se mezclaba entre situaciones normales y situaciones de error (todo con `if`)
- No existía propagación automática de errores

Ventajas de las excepciones

- Separación del código normal del código excepcional de manejo de errores
- Las cabeceras de los métodos son más naturales, los errores se declaran explícitamente (**throws**)
- Propagación automática de los errores si no los queremos capturar
- Agrupación y distinción de errores en jerarquías de herencia

Ventajas de las excepciones

- Todas las excepciones se pueden capturar, incluso las que evidencian errores de programación
 - `NullPointerException`
 - `ArrayIndexOutOfBoundsException`
 - `ClassCastException`
- Esta característica es muy útil cuando hacemos programas basados en plugins o módulos desarrollados por terceros para no detener la ejecución cuando se producen errores en los módulos